Cambridge University Press
978-0-521-67595-6 — Writing Scientific Software
Suely Oliveira , David E. Stewart
Frontmatter
More Information

WRITING SCIENTIFIC SOFTWARE:A GUIDE TO GOOD STYLE

The core of scientific computing is designing, writing, testing, debugging and modifying numerical software for application to a vast range of areas: from graphics, weather forecasting, and chemistry to engineering, biology, and finance. Scientists, engineers, and computer scientists need to write good, clear code, for speed, clarity, flexibility, and ease of re-use.

Oliveira and Stewart provide here a guide to writing numerical software, pointing out good practices to follow, and pitfalls to avoid. By following their advice, the reader will learn how to write efficient software, and how to test it for bugs, accuracy, and performance. Techniques are explained with a variety of programming languages, and illustrated with two extensive design examples, one in Fortran 90 and one in C++, along with other examples in C, C++, Fortran 90 and Java scattered throughout the book.

Common issues in numerical computing are dealt with: for example, whether to allocate or pass "scratch" memory for temporary use, how to pass parameters to a function that is itself passed to a routine, how to allocate multidimensional arrays in C/C++/Java, and how to create suitable interfaces for routines and libraries. Advanced topics, such as recursive data structures, template programming and type binders for numerical computing, blocking and unrolling loops for efficiency, how to design software for deep memory hierarchies, and amortized doubling for efficient memory use, are also included.

This manual of scientific computing style will prove to be an essential addition to the bookshelf and lab of everyone who writes numerical software.

WRITING SCIENTIFIC SOFTWARE: A GUIDE FOR GOOD STYLE

SUELY OLIVEIRA AND DAVID E. STEWART
University of Iowa

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

4843/24, 2nd Floor, Ansari Road, Daryaganj, Delhi - 110002, India

79 Anson Road, #06-04/06, Singapore 079906

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9780521675956

First published 2006

A catalogue record for this publication is available from the British Library

ISBN 978-0-521-85896-0 Hardback
ISBN 978-0-521-67595-6 Paperback

Contents

Cambridge University Press
978-0-521-67595-6 — Writing Scientific Software
Suely Oliveira , David E. Stewart
Frontmatter
More Information

Cambridge University Press
978-0-521-67595-6 — Writing Scientific Software
Suely Oliveira , David E. Stewart
Frontmatter
More Information

Preface

Mathematical algorithms, though usually invisible, are all around us. The microcomputer in your car controlling the fuel ignition uses a control algorithm embodying mathematical theories of dynamical systems; a Web search engine might use large-scale matrix computations; a "smart map" using a Global Positioning System to tell where you are and the best way to get home embodies numerous numerical and non-numerical algorithms; the design of modern aircraft involves simulating the aerodynamic and structural characteristics on powerful computers including supercomputers.

Behind these applications is software that does numerical computations. Often it is called scientific software, or engineering software; this software uses finite-precision floating-point (and occasionally fixed-point) numbers to represent continuous quantities.

If you are involved in writing software that does numerical computations, this book is for you. In it we try to provide tools for writing effective and efficient numerical software. If you are a numerical analyst, this book may open your eyes to software issues and techniques that are new to you. If you are a programmer, this book will explain pitfalls to avoid with floating-point arithmetic and how to get good performance without losing modern design techniques (or programming in Fortran 66). People in other areas with computing projects that involve significant numerical computation can find a bounty of useful information and techniques in this book.

But this is not a book of numerical recipes, or even a textbook for numerical analysis (numerical analysis being the study of mathematical algorithms and their behavior with finite precision floating-point arithmetic or other sources of computational errors). Nor is it a handbook on software development. It is about the development of a particular kind of software: numerical software. Several things make this kind of software a little different from other kinds of software:

- *It involves computations with floating-point numbers*. All computations with floating point arithmetic are necessarily approximate. How good are the approximations? That is the subject matter of numerical analysis. Proofs of correctness of algorithms can be irrelevant because either: (a) they completely ignore the effects of roundoff error, and so cannot identify numerical difficulties; or (b) they assume only exact properties of floating point arithmetic (floating point arithmetic is commutative $x + y = y + x$, but not associative $(x + y) + z \neq x + (y + z)$). In the latter case, they cannot prove anything useful about algorithms which are numerically accurate, but not exact (which is almost all of them).
- *It involves large-scale computations*. Large-scale computations can involve computing millions of quantities. Here efficiency is of critical importance, both in time and memory. While correctness is vital, efficiency has a special place in scientific computing. Programmers who wish to get the most out of their machine had better understand just how the hardware (and software) behind their compilers and operating systems work.
- *Requirements change rapidly*. Frequent changes in requirements and methods are a fact of life for scientific software, whether in a commercial or research environment. This means that the code had better be flexible or it will be scrapped and we will be programming from scratch again.

In every decade since the 1950s, the complexity of scientific software has increased a great deal. Object-oriented software has come to the fore in scientific and engineering software with the development of a plethora of object-oriented matrix libraries and finite element packages. Fortran used to be the clear language of choice for scientific software. That has changed. Much scientific software is now written in C, C++, Java, Matlab, Ada, and languages other than Fortran. Fortran has also changed. The Fortran 90 standard and the standards that have followed have pushed Fortran forward with many modern programming structures. But, many people who were educated on Fortran 77 or earlier versions of Fortran are unaware of these powerful new features, and of how they can be used to facilitate large-scale scientific software development. In this book when we refer to "Fortran" we will mean Fortran 90 unless another version is explicitly mentioned.

We have focused on C, C++ and Fortran 90 as the languages we know best, and are in greatest use for scientific and engineering computing. But we will also have things to say about using other languages for scientific computing, especially Java. This is not to say that other languages are not appropriate. One of the points we want to make is that many of the lessons learnt in one language can carry over to other languages, and help us to better understand the trade-offs involved in the choice of programming language and software design.

Occasionally we make historical notes about how certain systems and programming languages developed. Often important insights into the character of operating systems and programming languages, and how they are used, can be gleaned from the history of their development. It is also a useful reminder that the world of software (including scientific software) is not static – it is changing and changing rapidly. And often our best guide to the future is a look at the past – it is certainly a good antidote to the impression often given that programming languages are eternal.

This book has been divided into five parts. The first is about what scientific and engineering software is all about, and what makes it different from conventional software design: the approximations inherent in floating-point arithmetic and other aspects of scientific software make some approaches to software development of limited applicability or irrelevant. Instead, we need to understand numerical issues much better than with other kinds of software. Also, scientific software often has much more of an emphasis on performance – there is a real need for speed. But this must be tempered by the need to keep the structure of the code from becoming "fossilized", trying to maximize performance for some particular system. Instead we advocate a balance between flexibility and performance.

The second part is about software design. After a look at how things happen (how CPUs work, stacks and registers, variable allocation, compilers, linkers and interpreters), we emphasize practical software design and development techniques. These include incremental testing alongside some of the more practical of the "proof of correctness" ideas.

The third part is on efficiency – in both time and memory. To do this well requires a good understanding of both algorithms and computer architecture. The importance of locality is particularly emphasized. There is also a considerable amount on how to use dynamic memory allocation. This may be particularly useful for Fortran programmers who have so far avoided dynamic memory allocation.

Part IV is on tools for software development including online sources of scientific software, debuggers, and tools that have originated from the Unix operating system and have spread to many other environments.

Part V emphasizes the practicalities involved in programming scientific software. We have developed two medium-sized examples of numerical software development. One is a cubic spline library for constructing and evaluating various kinds of splines. The other is a multigrid system for the efficient iterative solution of large, sparse linear systems of equations. In these examples, the reader will see the issues discussed earlier in the context of some real examples.

As Isaac Newton said, "If I have seen far, it is because I have stood on the shoulders of giants." We do not claim to see as far as Isaac Newton, but we have stood on the shoulders of giants. We would like to thank Barry Smith, Michael Overton, Nicholas Higham, Kendall Atkinson, and the copy-editor for their comments on our manuscript. We would especially like to thank Cambridge University Press' technical reviewer, who was most assiduous in going through the manuscript, and whose many comments have resulted in a greatly improved manuscript. We would also like to thank the many sources of the software that we have used in the

production of this book: Microsoft (MS) Windows XP, Red-Hat Linux, the GNU compiler collection (`gcc`, `g++`, and most recently `g95`), Delorie's port of `gcc/g++` to MS Windows (`djgpp`), Minimal GNU for Windows (MinGW) and their port of `gcc/g++` to MS Windows, Intel's Fortran 90/95 compiler, the GNU tools `gmake`, `grep`, `sed`, `gdb`, etc. (many thanks to the Free Software Foundation for making these tools widely available), the LyX word processing software, the MikTeX and TeTeX implementations of LaTeX and the DVI viewers `xdvi` and `yap`, Component Software's implementation of RCS for MS Windows, Xfig, zip and unzip, WinZip, Valgrind, Octave and MATLAB.